*EXCEL*lent Office Adventures

David Shank
Microsoft Corporation

January 4, 2001

It is the first week of the new year and no matter how we spent our holidays, most of us have at least two things in common this week: We are back in the Office (as I like to say in this column) and we are trying to figure out how to pay for all the gifts and celebrations of last month.

For some of us, when it comes to number crunching, it is visions of Microsoft Excel that dance in our heads. And since I have not talked about working with Excel in this column, as many of you have reminded me, I've decided that now is the perfect time.

This is the first part of a two-part discussion on working with Excel. In this monthÃ‚'s column, I will discuss the basics of what you need to know to work with Excel programmatically. Next month, I will delve in a bit deeper to discuss working with ranges, regions, and cells.

The Excel Application Object

The Excel **Application** object is the top-level object in Excel's object model. You use the **Application** object to determine or specify application-level properties or execute application-level methods. The **Application** object is also the entry point into the rest of the Excel object model.

When you work with properties and methods of the **Application** object by using Visual Basic for Applications (VBA) from within Excel, the **Application** object is available to you by default. This is known as an implicit reference to the object. To work with Excel objects from another Office application, you must first create an object variable that represents the Excel **Application** object. This is known as an explicit reference to the object. For example, the following two procedures return the name of the currently active **Worksheet** object. The **ShowNameFromInsideXL** procedure is designed to work from within Excel and uses an implicit reference to the **Application** object. In other words, it references the **ActiveSheet** property of the **Application** object without explicitly referencing the **Application** object itself. You run the **ShowNameFromOutsideXL** procedure from outside Excel and so must use an explicit reference to the **Application** object.

```
Sub ShowNameFromInsideXL()
    MsgBox "'" & ActiveSheet.Name & "' is the active worksheet."
End Sub

Sub ShowNameFromOutsideXL()
    Dim xlApp As Excel.Application

    Const XL_NOTRUNNING As Long = 429

    On Error GoTo ShowName_Err
    Set xlApp = GetObject(, "Excel.Application")
    MsgBox "'" & xlApp.ActiveSheet.Name & "' is active."
    xlApp.Quit
    Set xlApp = Nothing

ShowName_End:
    Exit Sub
ShowName_Err:
    If Err = XL_NOTRUNNING Then
        ' Excel is not currently running.
        Set xlApp = New Excel.Application
        xlApp.Workbooks.Add
        Resume Next
    Else
        MsgBox Err.Number & " - " & Err.Description
    End If
```

```
    Resume ShowName_End
End Sub
```

Notice that the **ShowNameFromOutsideXL** procedure uses the **GetObject** function to get a reference to the currently running instance of Excel. If Excel is not running when this procedure is called, an error occurs. The error handler uses the **New** keyword to create a new instance of Excel, then adds a new **Workbook** object. Since a workbook will contain at least one worksheet, the remaining code in the procedure will execute correctly.

Note: To use the **New** keyword, you must have a reference set to the Excel object model from the project that contains the **ShowNameFromOutsideXL** procedure.

Understanding ExcelÃ‚Â's shortcuts to active objects

Like other Office application object models, the Excel **Application** object exposes several properties you can use to work with a currently active Excel object. For example, you will often write VBA procedures designed to work with information in the currently selected cell, or with the currently active worksheet. The **Application** object exposes the **ActiveCell**, **ActiveChart**, **ActivePrinter**, **ActiveSheet**, **ActiveWindow**, and **ActiveWorkbook** properties, which you can use to return a reference to the currently active cell, chart, printer, sheet, window, or workbook. The following examples illustrate various ways you might use some of these properties:

```
' ActiveWorkbook property example:
Function SaveBookAs(strFileName As String) As Boolean
    ActiveWorkbook.SaveAs ActiveWorkbook.Path & "\" & strFileName
End Function

' ActiveCell property example:
Function CustomFormatCell()
    With ActiveCell
        If IsNumeric(.Text) And .Formula < 0 Then
            With .Font
            .Bold = True
            .Italic = True
            .Borders.Color = 255
        End If
    End With
End Function

' ActiveSheet property example:
Function ChangeName(strNewName As String) As Boolean
    ActiveSheet.Name = strNewName
End Function
```

In addition to the **ActiveWorkbook** property, you can use the **Application** object's **Workbooks** and **Worksheets** properties to return equivalent Excel objects. The **Workbooks** property returns the **Workbooks** collection that contains all the currently open **Workbook** objects. The **Worksheets** property returns the **Sheets** collection associated with the currently active workbook. The following example uses the **Workbooks** property to determine if a workbook is already open, and if not, to open it:

```
Function OpenBook(strFilePath As String) As Boolean
    ' This procedure checks to see if the workbook
    ' specified in the strFilePath argument is open.
    ' If it is open, the workbook is activated. If it is
    ' not open, the procedure opens it.
    Dim wkbCurrent       As Excel.Workbook
    Dim strBookName      As String

    On Error GoTo OpenBook_Err

    ' Determine the name portion of the strFilePath argument.
    strBookName = NameFromPath(strFilePath)
    If Len(strBookName) = 0 Then Exit Function
    If Workbooks.Count > 0 Then
        For Each wkbCurrent In Workbooks
            If UCase$(wkbCurrent.Name) = UCase$(strBookName) Then
```

```
                wkbCurrent.Activate
                Exit Function
            End If
        Next wkbCurrent
    End If
    Workbooks.Open strBookName
    OpenBook = True

OpenBook_End:
    Exit Function
OpenBook_Err:
    OpenBook = False
    Resume OpenBook_End
End Function
```

In the preceding example, the **OpenBook** procedure calls a custom procedure named **NameFromPath** that returns the file name portion of the full path and file name passed to the **OpenBook** procedure in the **strFilePath** argument:

```
Function NameFromPath(strPath As String) As String
    ' This procedure takes a file path and returns
    ' the file name portion.
    Dim lngPos          As Long
    Dim strPart         As String
    Dim blnIncludesFile As Boolean

    ' Check that this is a file path.
    ' Find the last path separator.
    lngPos = InStrRev(strPath, "\")
    ' Determine if string after last backslash
    ' contains a period.
    blnIncludesFile = InStrRev(strPath, ".") > lngPos
    strPart = ""

    If lngPos > 0 Then
        If blnIncludesFile Then
            strPart = Right$(strPath, Len(strPath) - lngPos)
        End If
    End If
    NameFromPath = strPart
End Function
```

The Excel Workbook Object

In the Excel object model, the **Workbook** object appears just below the **Application** object. The **Workbook** object represents an Excel .xls or .xla workbook file. You use the **Workbook** object to work with a single Excel workbook. You use the **Workbooks** collection to work with all currently open **Workbook** objects.

You can also use the **Application** object's **ActiveWorkbook** property to return a reference to the currently active workbook. The **Workbooks** collection has a **Count** property you can use to determine how many visible and hidden workbooks are open. By default, Excel typically has one hidden workbook named Personal.xls. Excel uses the Personal.xls workbook as the default location to store macros. If the hidden Personal.xls workbook is the only open workbook, the **ActiveWorkbook** property returns **Nothing**, but the **Workbooks** collection's **Count** property returns 1. The **Workbooks** collection's **Count** property will return 0 only when there are no hidden or visible open workbooks.

Working with Workbook objects

You create a new **Workbook** object by using the **Workbooks** collection's **Add** method. The **Add** method not only creates a new workbook, but also immediately opens the workbook. The **Add** method also returns an object variable that represents the new workbook just created. The new workbook will contain the number of worksheets specified in the **Sheets in new workbook** box on the **General** tab of the **Options** dialog box (**Tools** menu). You can also specify the number of sheets a new workbook will have by using the **Application** object's **SheetsInNewWorkbook** property.

You can save a new workbook by using the **Workbook** object's **SaveAs** method and specifying the name of the workbook you want to save. If a workbook by that name already exists, an error occurs. Once a workbook has been saved by using the **SaveAs** method, additional changes are saved by using the **Workbook** object's **Save** method. You can also save a copy of an existing workbook with a different file name by using the **SaveCopyAs** method. You can supply a file name to be used with the **SaveAs** or **SaveCopyAs** method, or you can use the **Application** object's **GetSaveAsFileName** method to let the user supply the name to be used to save the workbook. If the user clicks **Cancel** in the **Save As** dialog box, the **GetSaveAsFileName** method returns **False.**

Before you save a new workbook by using the **SaveAs** method, the **Workbook** object's **Name** property setting is a value assigned by Excel, such as Book1.xls. After you save the workbook, the **Name** property contains the name you supplied in the **Filename** argument of the **SaveAs** method. The **Name** property is read-only; to change the name of a workbook, you must use the **SaveAs** method again and pass a different value in the **Filename** argument.

Note: A **Workbook** object's **FullName** property contains the object's path and file name, whereas the **Path** property contains only the saved path to the current workbook. Before a new workbook is saved, the **FullName** property has the same value as the **Name** property, and the **Path** property has no value.

The **Workbooks** collection's **Open** method opens an existing workbook. When you open a workbook by using the **Open** method, it also becomes the active workbook. You can supply a file name to be used with the **Open** method, or you can use the **Application** object's **GetOpenFileName** method to let the user select the workbook to open. If the user clicks **Cancel** in the **Open** dialog box, the **GetOpenFileName** method returns **False**.

You use a **Workbook** object's **Close** method to close an open workbook. To specify whether pending changes to the workbook should be saved before the object is closed, you use the **SaveChanges** argument. If the **SaveChanges** argument is omitted, the user is prompted to save pending changes. You can also use the **Close** method of the **Workbooks** object to close all open workbooks. If there are unsaved changes to any open workbook when this method is used, the user is prompted to save changes. If the user clicks **Cancel** in this **Save** dialog box, an error occurs that your code must handle. You can suppress this **Save** dialog box by setting the **Application** object's **DisplayAlerts** property to **False** before executing the **Close** method. When you use the **Workbooks** object's **Close** method in this manner, any unsaved changes to open workbooks are lost. After the **Close** method has run, remember to set the **DisplayAlerts** property to **True**.

Note: The **Auto_Open** and **Auto_Close** event procedures are ignored when a workbook is opened or closed by using the **Open** or **Close** methods. You can force these procedures to run by using the **Workbook** object's **RunAutoMacros** method. The VBA code in a workbook's **Open** and **BeforeClose** event procedures will be executed when the workbook is opened or closed by using the **Open** or **Close** methods.

The following example illustrates how to create a new workbook and specify the number of worksheets it will have:

```
Function CreateNewWorkbook(Optional strBookName As String = "", _
         Optional intNumSheets As Integer = 3) As Workbook
    ' This procedure creates a new workbook file and
    ' saves it by using the path and name specified
    ' in the strBookName argument. You use the intNumsheets
    ' argument to specify the number of worksheets in the
    ' workbook; the default is three.

    Dim intOrigNumSheets      As Integer
    Dim wkbNew                As Excel.Workbook

    On Error GoTo CreateNew_Err

    intOrigNumSheets = Application.SheetsInNewWorkbook
    If intOrigNumSheets <> intNumSheets Then
        Application.SheetsInNewWorkbook = intNumSheets
    End If
    Set wkbNew = Workbooks.Add
    If Len(strBookName) = 0 Then _
        strBookName = Application.GetSaveAsFilename
    wkbNew.SaveAs strBookName
```

```
    Set CreateNewWorkbook = wkbNew
    Application.SheetsInNewWorkbook = intOrigNumSheets

CreateNew_End:
    Exit Function
CreateNew_Err:
    Set CreateNewWorkbook = Nothing
    wkbNew.Close False
    Set wkbNew = Nothing
    Resume CreateNew_End
End Function
```

A **Workbook** object's **Saved** property is a **Boolean** value that indicates whether the workbook has been saved. The **Saved** property will be **True** for any new or opened workbook where no changes have been made and **False** for a workbook that has unsaved changes. You can set the **Saved** property to **True**. Doing this prevents the user from being prompted to save changes when the workbook closes but does not actually save any changes made since the last time the workbook was saved by using the **Save** method.

A note about working with workbooks through automation

When you are using Automation to edit an Excel workbook, keep the following in mind.

Creating a new instance of Excel and opening a workbook results in an invisible instance of Excel and a hidden instance of the workbook. Therefore, if you edit the workbook and save it, the workbook is saved as hidden. The next time the user opens Excel manually, the workbook is invisible and the user has to click **Unhide** on the **Window** menu to view the workbook.

To avoid this behavior, your Automation code should unhide the workbook before editing it and saving it. Note that this does *not* mean that Excel itself has to be visible.

The Excel Worksheet Object

Most of the work you will do in Excel will be within the context of a worksheet. A worksheet contains a grid of cells you can use to work with data, and hundreds of properties, methods, and events you can use to work with the data in a worksheet.

To work with the data contained in a worksheet, in a cell or within a range of cells, you use a **Range** object. The **Worksheet** and **Range** objects are the two most basic and most important components of any custom solution you create within Excel. I will talk more about the Range object in next monthÃ‚Â's column.

The **Workbook** object's **Worksheets** property returns a collection of all the worksheets in the workbook. The **Workbook** object's **Sheets** property returns a collection of all the worksheets and chart sheets in the workbook.

Each Excel workbook contains one or more **Worksheet** objects and can contain one or more chart sheets as well. Charts in Excel are either embedded in a worksheet or contained on a chart sheet. You can have only one chart on a chart sheet, but you can have multiple charts on a worksheet. Each embedded chart on a worksheet is a member of the **Worksheet** object's **ChartObjects** collection. **Worksheet** objects are contained in the **Worksheets** collection, which you can access by using the **Workbook** object's **Worksheets** property. When you use VBA to create a new workbook, you can specify how many worksheets it will contain by using the **Application** object's **SheetsInNewWorkbook** property.

Referring to a worksheet

Because a **Worksheet** object exists as a member of a **Worksheets** collection, you refer to a worksheet by its name or its index value. In the following example, both object variables refer to the first worksheet in a workbook:

```
Sub ReferToWorksheetExample()
    ' This procedure illustrates how to programmatically refer to
    ' a worksheet.
    Dim wksSheetByIndex     As Excel.Worksheet
    Dim wksSheetByName      As Excel.Worksheet
```

```
    With ActiveWorkbook
        Set wksSheetByIndex = Worksheets(1)
        Set wksSheetByName = Worksheets("Main")
        If wksSheetByIndex.Index = wksSheetByName.Index Then
            MsgBox "The worksheet indexed as #" _
                & wksSheetByIndex.Index & vbCrLf _
                & "is the same as the worksheet named '" _
                & wksSheetByName.Name & "'", _
                vbOKOnly, "Worksheets Match!"
        End If
    End With
End Sub
```

Working with a worksheet

You can add one or more worksheets to the **Worksheets** collection by using the collection's **Add** method. The **Add** method returns the new **Worksheet** object. If you add multiple worksheets, the **Add** method returns the last worksheet added to the **Worksheets** collection. If the **Before** or **After** arguments of the **Add** method are omitted, the new worksheet is added before the currently active worksheet. The following example adds a new worksheet before the active worksheet in the current collection of worksheets:

```
Dim wksNewSheet As Excel.Worksheet

Set wksNewSheet = Worksheets.Add
With wksNewSheet
    ' Work with properties and methods of the
    ' new worksheet here.
End With
```

You use the **Worksheet** object's **Delete** method to delete a worksheet from the **Worksheets** collection. When you try to programmatically delete a worksheet, Excel will display a message (alert); to suppress the message, you must set the **Application** object's **DisplayAlerts** property to **False**, as illustrated in the following example:

```
Function DeleteWorksheet(strSheetName As String) As Boolean
    On Error Resume Next

    Application.DisplayAlerts = False
    ActiveWorkbook.Worksheets(strSheetName).Delete
    Application.DisplayAlerts = True
    ' Return True if no error occurred;
    ' otherwise return False.
    DeleteWorksheet = Not CBool(Err.Number)
End Function
```

Note: When you set the **DisplayAlerts** property to **False**, always set it back to **True** before your procedure has finished executing, as shown in the preceding example.

You can copy a worksheet by using the **Worksheet** object's **Copy** method. To copy a worksheet to the same workbook as the source worksheet, you must specify either the **Before** or **After** argument of the **Copy** method. You move a worksheet by using the **Worksheet** object's **Move** method. For example:

```
Worksheets("Sheet1").Copy After:=Worksheets("Sheet3")
Worksheets("Sheet1").Move After:=Worksheets("Sheet3")
```

The next example illustrates how to move a worksheet so that it is the last worksheet in a workbook:

```
Worksheets("Sheet1").Move After:=Worksheets(Worksheets.Count)
```

Note: When you use either the **Copy** or the **Move** method, if you do not specify the **Before** or **After** argument, Excel creates a new workbook and copies the specified worksheet to it.

Where to Get More Info

The techniques discussed in this column will get you started working with Excel programmatically. For additional information, check out the following resources:

- For more information on the Excel object model see the Microsoft Excel Object Model.
- As always, check in regularly at the Office Developer Center for information and technical articles on Office solution development.

David Shank *is a programmer/writer on the Office team specializing in developer documentation. Rumor has it he lives high in the mountains to the east of Redmond and is one of the few native Northwesterners who still lives in the Northwest.*

*EXCEL*lent Office Adventures, Part 2

David Shank
Microsoft Corporation

February 1, 2001

In last monthÃ‚Â's column, I provided an overview of what you need to know to work with the Excel object model. This month I will build on that discussion with information on working with the **Range** object and some of its properties and methods.

Understanding the Range Object

ExcelÃ‚Â's **Range** object is one of the most powerful, dynamic, and often-used objects in the Excel object model. Once you develop a full understanding of the **Range** object and how to use it effectively in Visual Basic for Applications (VBA) procedures, you will be well on your way to programmatically harnessing the power of Excel.

The Excel **Range** object is unique among objects. In most cases, an object is a thing with some clearly identifiable corollary in the Excel user interface. For example, a **Workbook** object is recognizable as an .xls file. In a workbook, the collection of **Worksheet** objects is represented in the user interface by separate tabbed sheets. But the **Range** object is different. A range can be a different thing in different circumstances. A **Range** object can be a single cell or a collection of cells. It can be a single object or a collection of objects. It can be a row or column, or it can represent a three-dimensional collection of cells that span multiple worksheets. In addition, unlike other objects that exist as objects and as members of a collection of objects, there is no **Ranges** collection that contains all **Range** objects in a workbook or worksheet. It is probably easiest to think of the **Range** object as your handle to the thing you want to work with.

Because the **Range** object is such a fundamental entity within Excel, you will find that many properties and methods return a **Range** object that you can use to work with the data in your custom solution. The following sections discuss some basic aspects of **Range** objects and many of the ways you can return a **Range** object from a built-in property or method.

Working with the Range property

You will use the **Range** property to return a **Range** object in many different circumstances. The **Application** object, the **Worksheet** object, and the **Range** object all have a **Range** property. The **Application** object's **Range** property returns the same **Range** object as that returned by the **Worksheet** object. In other words, the **Application** object's **Range** property returns a reference to the specified cell or cells on the active worksheet. The **Range** property of the **Range** object has a subtle difference that it is important to understand. Consider the following example:

```
Dim rng1 As Range
Dim rng2 As Range
Dim rng3 As Range

Set rng1 = Application.Range("B5")
Set rng2 = Worksheets("Sheet1").Range("B5")
Set rng3 = rng2.Range("B5")
```

These three **Range** objects do not all return a reference to the same cell. In this example, **rng1** and **rng2** both return a reference to cell B5. But **rng3** returns a reference to cell C9. This difference occurs because the **Range** object's **Range** property returns a reference *relative* to the specified cell. In this case, the specified cell is B5. Therefore, the B means that the reference will be one column to the right of B5, and the 5 means the reference will be the fifth row below the row specified by B5, including the current (fifth) row. In other words, the **Range** object's **Range** property returns a reference to a cell that is *n* columns to the right and *y* rows down from the specified cell).

Typically, you will use the **Range** property to return a **Range** object, then use the properties and methods of that

Range object to work with the data in a cell or group of cells. The following table contains several examples that illustrate how you might work with the **Range** property:

Description	Range Property Example
Set the value of cell A1 on Sheet1 to 100	Worksheets("Sheet1").Range("A1").Value = 100
Set the value for a group of cells on the active worksheet	Range("B2:B14").Value = 10000
Set the formula for cell B15 on the active worksheet	Range("B15").Formula = "=Sum(B2:B14)"
Set the font to bold	Range("B15").Font.Bold = True
Set the font color to green	Range("B15").Font.Color = RGB(0, 255, 0)
Set an object variable to refer to a single cell	Set rngCurrent = Range("A1")
Set an object variable to refer to a group of cells	Set rngCurrent = Range("A1:L1")
Format all the cells in a named range	Range("YTDSalesTotals").Font.Bold = True
Set an object variable to a named range	Set rngCurrent = Range("NovemberReturns")
Set an object variable representing all the used cells on the Employees worksheet	Set rngCurrent = Worksheets("Employees").UsedRange
Set an object variable representing the group of related cells that surround the active cell	Set rngCurrent = ActiveCell.CurrentRegion
Set an object variable representing the first three columns in the active worksheet	Set rngCurrent = Range("A:C")
Set an object variable representing rows 3, 5, 7, and 9 of the active worksheet	Set rngCurrent = Range("3:3, 5:5, 7:7, 9:9")
Set an object variable representing multiple noncontiguous groups of cells on the active sheet	Set rngCurrent = Range("A1:C4, D6:G12, I2:L7")
Remove the contents for all cells within a specified group of cells (B5:B10) while leaving the formatting intact	Range("B5", "B10").ClearContents

As you can see from the examples, the **Cell** argument of the **Range** property can be either an A1-style string reference or a string that represents a named range within the current workbook.

You can also use the **Range** property to return **Range** objects in arguments for other methods in the Excel object model. When you use the **Range** property in this way, make sure you fully qualify the **Worksheet** object to which the **Range** property applies. Failing to use fully qualified references to the **Range** property in arguments for Excel methods is one of the most common sources of error in range-related code.

Working with the active cell or the current selection

The **ActiveCell** property returns a **Range** object that represents the currently active cell. When a single cell is selected, the **ActiveCell** property returns a **Range** object that represents that single cell. When multiple cells are selected, the **ActiveCell** property represents the single active cell within the current selection. When a cell or group of cells is selected, the **Selection** property returns a **Range** object representing all the cells within the current selection.

To understand how the **ActiveCell** and **Selection** properties relate, consider a case in which a user selects cells A1 through F1 by clicking cell A1 and dragging until the selection extends over cell F1. In this case, the **ActiveCell**

property returns a **Range** object that represents cell A1. The **Selection** property returns a **Range** object that represents cells A1 through F1.

When you work with Excel's user interface, you typically select a cell or group of cells and then perform an action on the selected cell or cells, such as entering a value for a single cell or formatting a group of cells. When you use VBA to work with cells, you don't need to make a selection before performing an action on a cell or group of cells. Instead, you need only return a **Range** object that represents the cell or cells you want to work with. For example, to enter January as the value for cell A1 by using the user interface, you would select cell A1 and type **January**. The following sample performs the same action in VBA:

```
ActiveSheet.Range("A1").Value = "January"
```

Using VBA to work with a **Range** object in this manner does not change the selected cells on the current worksheet. However, you can make your VBA code act upon cells in the same way as a user working through the user interface by using the **Range** object's **Select** method to select a cell or range of cells, then using the **Range** object's **Activate** method to activate a cell within the current selection. For example, the following code selects cells A1 through A6, then makes cell A3 the active cell:

```
With ActiveSheet
    .Range("A1:A6").Select
    .Range("A3").Activate
End With
```

When you use the **Select** method to select multiple cells, the first cell referenced will be the active cell. For example, in the preceding sample, after the **Select** method is executed, the **ActiveCell** property returns a reference to cell A1, even though cells A1 through A6 are selected. After the **Activate** method is executed in the next line of code, the **ActiveCell** property returns a reference to cell A3 while cells A1 through A6 remain selected. The next example illustrates how to return a **Range** object by using the **ActiveCell** property or the **Selection** property:

```
Dim rngActiveRange As Excel.Range

' Range object returned from the Selection property.
Set rngActiveRange = Selection
Call PrintRangeInfo(rngActiveRange)
' Range object returned from the ActiveCell property.
Set rngActiveRange = ActiveCell
Call PrintRangeInfo(rngActiveRange)
```

The **PrintRangeInfo** custom procedure called in the preceding example prints information about the cell or cells contained in the **Range** object passed in the argument to the procedure.

```
Sub PrintRangeInfo(rngCurrent As Excel.Range)
    Dim rngTemp         As Excel.Range
    Dim strValue        As String
    Dim strRangeName    As String
    Dim strAddress      As String
    Dim strFormula      As String

    On Error Resume Next

    strRangeName = rngCurrent.Name.Name _
        & " (" & rngCurrent.Name.RefersTo & ")"
    strAddress = rngCurrent.Address

    For Each rngTemp In rngCurrent.Cells
        If IsEmpty(rngTemp) Then
            strValue = strValue & "Cell(" & rngTemp.Address _
                & ") Is empty." & vbCrLf
        Else
            strValue = strValue & "Cell(" & rngTemp.Address _
                & ") = " & rngTemp.Value _
                & " Formula " & rngTemp.Formula & vbCrLf
```

```
        End If
    Next rngTemp
    Debug.Print IIf(Len(strRangeName) > 0, "Range Name = " _
        & strRangeName, "Range not named") & vbCrLf & "Address = " _
        & strAddress & vbCrLf & strValue
    Debug.Print "**********************************"
    Debug.Print
    If Err > 0 Then Err = 0
End Sub
```

Working with cells and groups of cells

You will often have to write code to work against a range of cells, but at the time you write the code you will not have information about the range. For example, you may not know the size or location of a range or the location of a cell in relation to another cell. You can use the **CurrentRegion** and **UsedRange** properties to work with a range of cells whose size you have no control over. You can use the **Offset** property to work with cells in relation to other cells where the cell location is unknown.

The **Range** object's **CurrentRegion** property returns a **Range** object that represents a range bounded by (but not including) any combination of blank rows and blank columns or the edges of the worksheet. For example, if you had a table of data within cells D3 through E12 and the focus was in cell D3, then the **CurrentRegion** property would return a **Range** object that represents cells D3 through E12.

The **CurrentRegion** property can return many ranges on a single worksheet. This property is useful for operations where you need to know the dimensions of a group of related cells, but all you know for sure is the location of a cell or cells within the group. For example, if the active cell were inside a table of cells, you could use the following line of code to apply formatting to the entire table:

```
ActiveCell.CurrentRegion.AutoFormat xlRangeAutoFormatAccounting4
```

You could also use the **CurrentRegion** property to return a collection of cells. For example:

```
Dim rngCurrentCell As Excel.Range

For Each rngCurrentCell In ActiveCell.CurrentRegion.Cells
    ' Work with individual cells here.
Next rngCurrentCell
```

Every **Worksheet** object has a **UsedRange** property that returns a **Range** object representing the area of a worksheet that is being used. The **UsedRange** property represents the area described by the farthest upper-left and farthest lower-right cells that contain data on a worksheet, and includes all cells in between regardless of whether they contain data. For example, imagine a worksheet with entries in only two cells: A1 and G55. The worksheet's **UsedRange** property would return a **Range** object that contains all the cells from A1 to G55.

You might use the **UsedRange** property together with the **SpecialCells** method to return a **Range** object that represents all cells of a specified type on a worksheet. For example, the following code returns a **Range** object that includes all the cells in the active worksheet that contain a formula:

```
Dim rngFormulas As Excel.Range
Set rngFormulas = ActiveSheet.UsedRange.SpecialCells(xlCellTypeFormulas)
```

You can use the **Cells** property to loop through a range of cells on a worksheet or to refer to a range by using numeric row and column values. The **Cells** property returns a **Range** object that represents all the cells, or a specified cell, in a worksheet. To work with a single cell, you use the **Item** property of the **Range** object returned by the **Cells** property, and specify the index of a specific cell you want to work with. The **Item** property accepts arguments that specify the row or the row and column index for a cell.

Since the **Item** property is the default property of the **Range** object, it is not necessary to explicitly reference it. For example, the following **Set** statements both return a reference to cell B5 on Sheet1:

```
Dim rng1 As Excel.Range
Dim rng2 As Excel.Range

Set rng1 = Worksheet("Sheet1").Cells.Item(5, 2)
Set rng2 = Worksheet("Sheet1").Cells(5, 2)
```

The row and column index arguments of the **Item** property return references to individual cells, beginning with the first cell in the specified range. For example, the following message box displays G11 because that is the first cell in the specified **Range** object:

```
MsgBox Range("G11:M30").Cells(1,1).Address
```

The following procedure illustrates how you would use the **Cells** property to loop through all the cells in a specified range. The **OutOfBounds** procedure looks for values that are greater than or less than a specified range of values, and changes the font color for each cell with such a value:

```
Function OutOfBounds(rngToCheck As Excel.Range, _
         lngLowValue As Long, _
         lngHighValue As Long, _
         Optional lngHighlightColor As Long = 255) As Boolean
    ' This procedure illustrates how to use the Cells property
    ' to iterate through a collection of cells in a range.
    ' For each cell in the rngTocheck range, if the value of the
    ' cell is numeric and it falls outside the range of values
    ' specified by lngLowValue to lngHighValue, the cell font
    ' is changed to the value of lngHighlightColor
    ' (default is red).
    Dim rngTemp           As Excel.Range
    Dim lngRowCounter     As Long
    Dim lngColCounter     As Long

    ' Validate bounds parameters.
    If lngLowValue > lngHighValue Then
         Err.Raise vbObjectError + 512 + 1, _
            "OutOfBounds Procedure", _
            "Invalid bounds parameters submitted: " _
            & "Low value must be lower than high value."
        Exit Function
    End If

    ' Iterate through cells and determine if values
    ' are outside bounds parameters. If so, highlight value.
    For lngRowCounter = 1 To rngToCheck.Rows.Count
        For lngColCounter = 1 To rngToCheck.Columns.Count
            Set rngTemp = rngToCheck.Cells(lngRowCounter, _
                 lngColCounter)
            If IsNumeric(rngTemp.Value) Then
                If rngTemp.Value < lngLowValue Or _
                    rngTemp.Value > lngHighValue Then
                    rngTemp.Font.Color = lngHighlightColor
                    OutOfBounds = True
                End If
            End If
        Next lngColCounter
    Next lngRowCounter
End Function
```

You can also use a **For EachÃ‚Â…Next** statement to loop through the range returned by the **Cells** property. The following code could be used in the **OutOfBounds** procedure to loop through cells in a range:

```
' Iterate through cells and determine if values
' are outside bounds parameters. If so, highlight value.
For Each rngTemp in rngToCheck.Cells
    If IsNumeric(rngTemp.Value) Then
        If rngTemp.Value < lngLowValue Or _
```

```
            rngTemp.Value > lngHighValue Then
            rngTemp.Font.Color = lngHighlightColor
            OutOfBounds = True
        End If
    End If
Next rngTemp
```

You can use the **Offset** property to return a **Range** object with the same dimensions as a specified **Range** object, but offset from the specified range. For example, you could use the **Offset** property to create a new **Range** object adjacent to the active cell to contain calculated values based on the active cell.

The **Offset** property is useful when you do not know the specific address of the cells you will need to work with, but you do know where the cell is located *in relation to* other cells you need to work with. For example, you may have a command bar button in your custom solution that fills the active cell with the average of the values in the two cells immediately to the left of the active cell:

```
ActiveCell.Value = (ActiveCell.Offset(0, -2) + ActiveCell.Offset(0, -1)/2)
```

Where to Get More Info

For additional information about working with Excel, check out the following resources:

- For an overview of working with the Excel object, see last month's column.
- For more information on the Excel object model see http://msdn.microsoft.com/library/default.asp?URL=/library/officedev/odeomg/deovrmicrosoftexcel2000.htm.
- For more information on working with Office application object models, see Understanding Office Objects and Object Models.
- As always, check in regularly at the Office Developer Center for information and technical articles on Office solution development.